Chapter 20
Class Inheritance

20.1 Introduction

Inheritance is a core feature of Object-Oriented Programming. It allows one class to *inherit* data or behaviour from another class and is one of the key ways in which reuse is enabled within classes.

This chapter introduces inheritance between classes in Python.

20.2 What Is Inheritance?

Inheritance allows features defined in one class to be *inherited* and reused in the definition of another class. For example, a `Person` class might have the attributes `name` and `age`. It might also have behaviour associated with a `Person` such as `birthday()`.

We might then decide that we want to have another class `Employee` and that employees also have a `name` and an `age` and will have birthdays. However, in addition an `Employee` may have an employee `Id` attribute and a `calculate_pay()` behaviour.

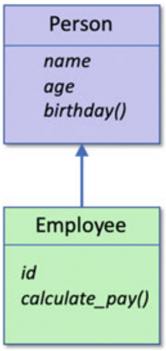
At this point we could duplicate the definition of the `name` and `age` attributes and the `birthday()` behaviour in the class `Employee` (for example by cutting and pasting the code between the two classes).

However, this is not only inefficient; it may also cause problems in the future. For example we may realise that there is a problem or bug in the implementation of `birthday()` and may correct it in the class `Person`; however, we may forget to apply the same fix to the class `Employee`.

In general, in software design and development it is considered best practice to define something once and to reuse that something when required.

J. Hunt, *A Beginners Guide to Python 3 Programming*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-030-20290-3_20

In an object-oriented system we can achieve the reuse of data or behaviour via inheritance. That is one class (in this case the `Employee` class) can *inherit* features from another class (in this case `Person`). This is shown pictorially below:



In this diagram the `Employee` class is shown as inheriting from the `Person` class. This means that the `Employee` class obtains all the data and behaviour of the `Person` class. It is therefore as though the `Employee` class has defined three attributes `name`, `age` and `id` and two methods `birthday()` and `calculate_pay()`.

A class that is defined as extending a parent class has the following syntax:

```
class SubClassName(BaseClassName):
    class-body
```

Note that the parent class is specified by providing the name of that class in round brackets after the name of the new (child) class.

We can define the class `Person` in Python as before:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def birthday(self):
        print('Happy birthday you were', self.age)
        self.age += 1
        print('You are now', self.age)
```

We could now define the class `Employee` as being a class whose definition builds on (or inherits from) the class `Person`:

```
class Employee(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.id = id

    def calculate_pay(self, hours_worked):
        rate_of_pay = 7.50
        if self.age >= 21:
            rate_of_pay += 2.50
        return hours_worked * rate_of_pay
```

Here we do several things:

1. The class is called Employee but it extends Person. This is indicated by including the name of the class being inherited in parentheses after the name of the class being defined (e.g. Employee(Person)) in the class declaration.
2. Inside the __init__ method we reference the __init__() method defined in the class Person and used to initialise instances of that class (via the super().__init__() reference. This allows whatever initialisation is required for Person to happen. This is called from within the Employee class's __init__() which then allows any initialisation required by the Employee to occur. Note that the call to the super().__init__() initialiser can come anywhere within the Employee.__init__() method; but by convention it comes first to ensure that whatever the Person class does during initialisation does not over write what happens in the Employee class.
3. All instances of the class Person have a name, and age and have the behaviour birthday().
4. All instances of the class Employee have a name, and age and an id and have the behaviours birthday() and calculate_pay(house_worked).
5. The method calculate_pay() defined in the Employee class can access the attributes name and age just as it can access the attribute id. In fact, it uses the employee's age to determine the rate of pay to apply.

We can go further, and we can subclass Employee, for example with the class SalesPerson:

```
class SalesPerson(Employee):
    def __init__(self, name, age, id, region, sales):
        super().__init__(name, age, id)
        self.region = region
        self.sales = sales

    def bonus(self):
        return self.sales * 0.5
```

Now we can say that the class SalesPerson has a name, an age and an id as well as a region and a sales total. It also has the methods birthday(), calculate_pay(hourse_worked) and bonus().

In this case the SalesPerson.__init__() method calls the Employee.__init__() method as that is the next class up the hierarchy and thus we want to run that classes initialisation behaviour before we set up the SalesPerson class (which of course in turn runs the Person classes initialisation behaviour).

We can now write code such as:

```
print('Person')
p = Person('John', 54)
print(p)
print('-' * 25)

print('Employee')
e = Employee('Denise', 51, 7468)
e.birthday()
print('e.calculate_pay(40):', e.calculate_pay(40))
print('-' * 25)

print('SalesPerson')
s = SalesPerson('Phoebe', 21, 4712, 'UK', 30000.0)
s.birthday()
print('s.calculate_pay(40):', s.calculate_pay(40))
print('s.bonus():', s.bonus())
```

With the output being:

```
Person
John is 54
-------------------------
Employee
Happy birthday you were 51
You are now 52
e.calculate_pay(40): 400.0
-------------------------
SalesPerson
Happy birthday you were 21
You are now 22
s.calculate_pay(40): 400.0
s.bonus(): 15000.0
```

It is important to note that we have not done anything to the class Person by defining Employee and SalesPerson; that is it is not affected by those class definitions. Thus, a Person *does not* have an employee id. Similarly, neither an Employee nor a Person have a region or a sales total.

In terms of behaviour, instances of all three classes can run the method `birthday()`, but

- only `Employee` and `SalesPerson` objects can run the method `calculcate_pay()` and
- only `SalesPerson` objects can run the method `bonus()`.

20.3 Terminology Around Inheritance

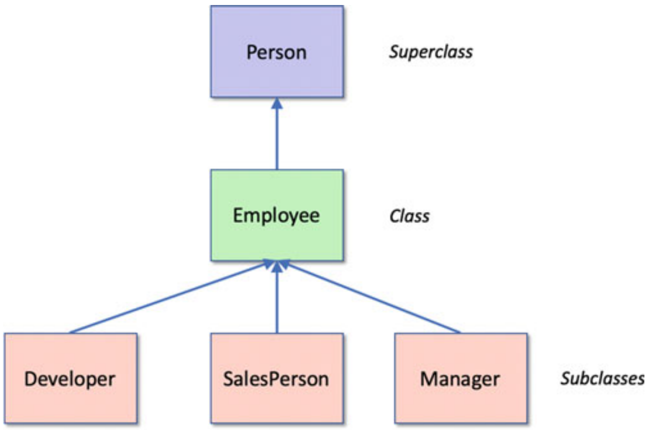
The following terminology is commonly used with inheritance in most object oriented languages including Python:

Class A class defines a combination of data and procedures that operate on that data.

Subclass A subclass is a class that inherits from another class. For example, an `Employee` might inherit from a class `Person`. Subclasses are, of course, classes in their own right. Any class can have any number of subclasses.

Superclass A superclass is the parent of a class. It is the class from which the current class inherits. For example, `Person` might be the superclass of `Employee`. In Python, a class can have any number of superclasses.

Single or multiple inheritance Single and multiple inheritance refer to the number of super classes from which a class can inherit. For example, Java is a single inheritance system, in which a class can only inherit from one class. Python by contrast is a multiple inheritance system in which a class can inherit from one or more classes.



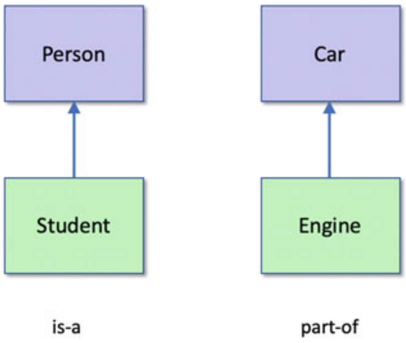
Note that a set of classes, involved in an inheritance hierarchy, such as those shown above, are often named after the class at the root (top) of the hierarchy; in this case it would make these classes part of the *Person* class hierarchy.

Types of Hierarchy

In most object-oriented systems there are two types of hierarchy; one refers to *inheritance* (whether single or multiple) and the other refers to *instantiation*. The inheritance hierarchy has already been described. It is the way in which one class inherits features from a superclass.

The instantiation hierarchy relates to instances or objects rather than classes and is important during the execution of the object.

There are two types of instance relationships: one indicates a *part-of* relationship, while the other relates to a *using* relationship (it is referred to as an *is-a* relationship). This is illustrated below:

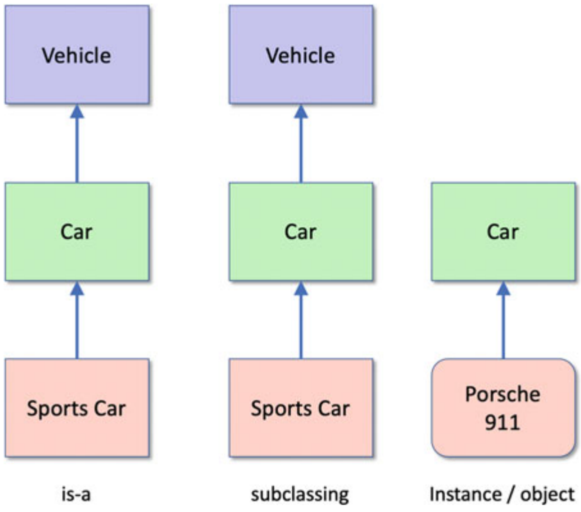


The difference between an *is-a* relationship and a *part-of* relationship is often confusing for new programmers (and sometimes for those who are experienced in non object oriented languages). The above figure illustrates that a Student *is-a* type of Person whereas an Engine is *part-of* a Car. It does not make sense to say that a student is part-of a person or that an engine is-a type of car!

In Python, *inheritance* relationships are implemented by the sub-classing mechanism. In contrast, part-of relationships are implemented using instance attributes in Python.

The problem with classes, inheritance and is-a relationships is that on the surface they appear to capture a similar concept. In the following figure the hierarchies all capture some aspect of the use of the phrase *is-a*. However, they are all intended to capture a different relationship.

The confusion is due to the fact that in modern English we tend to overuse the term *is-a*. For example, in English we can say that an Employee is a type of Person or that Andrew is a Person; both are semantically correct. However, in Python classes such as Employee and Person and an object such as Andrew are different things. We can distinguish between the different types of relationship by being more precise about our definitions in terms of a programming language, such as Python.



20.4 The Class Object and Inheritance

Every class in Python extends one or more superclasses. This is true even of the class `Person` shown below:

```
class Person:
   def __init__(self, name, age):
    self.name = name
    self.age = age
```

This is because if you do not specify a superclass explicitly Python automatically adds in the class object as a parent class. Thus the above is exactly the same as the following listing which explicitly lists the class `object` as the superclass of `Person`:

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Both listings above define a class called `Person` that extends the class `object`.

In fact, between Python 2.2 and Python 3 it was required to use the long hand form to ensure that the *new style* classes were being used (as opposed to an older way in which classes were defined pre Python 2.2). As such it is common to find that Python developers still use the long hand (explicit) form when defining classes that directly extend object.

The fact that all class eventually inherit from the class object means that behaviour defined in object is available for all classes everywhere.

20.5 The Built-in Object Class

The class object is the base (root) class for all classes in Python. It has methods that are therefore available in all Python objects. It defines a common set of *special* methods and *intrinsic* attributes. The methods include the special methods __str__(), __init()__, __eq__() (equals) and __hash__() (hash method). It also defines attributes such as __class__, __dict__, __doc__ and __module__.

20.6 Purpose of Subclasses

Subclasses are used to refine the behaviour and data structures of a superclass.

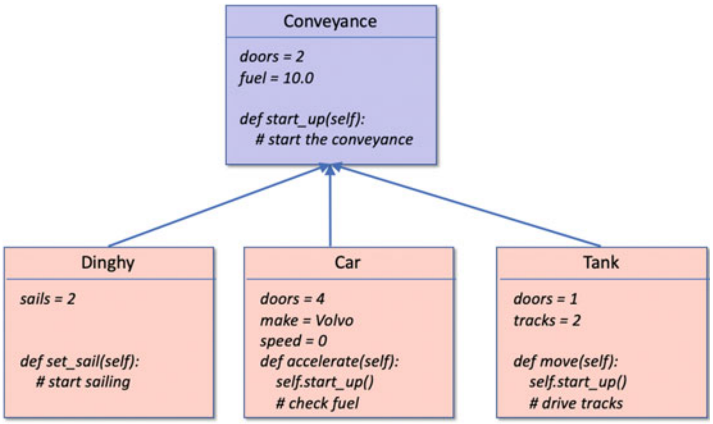
A parent class may define some generic/shared attributes and methods; these can then be inherited and reused by several other (sub) classes which add subclass specific attributes and behaviour.

In fact, there are only a small number of things that a subclass should do relative to its parent or super class. If a proposed subclass does not do any of these then your selected parent class is not the most appropriate super class to use.

A subclass should modify the behaviour of its parent class or extend the data held by its parent class. This modification should refine the class in one or more of these ways:

- Changes to the external protocol or interface of the class, that is it should extend the set of methods or attributes provided by the class.
- Changes in the implementation of the methods; i.e. the way in which the behaviour provided by the class are implemented.
- Additional behaviour that references inherited behaviour.

If a subclass does not provide one or more of the above, then it is incorrectly placed. For example, if a subclass implements a set of new methods, but does not refer to the attributes or methods of the parent class, then the class is not really a subclass of the parent (it does not extend it).



As an example, consider the class hierarchy illustrated above. A generic root class has been defined. This class defines a `Conveyance` which has doors, fuel (both with default values) and a method, `start_up()`, that starts the engine of the conveyance. Three subclasses of `Conveyance` have also been defined: `Dinghy`, `Car` and `Tank`. Two of these subclasses are appropriate, but one should probably not inherit from `Conveyance`. We shall consider each in turn to determine their suitability.

- The class `Tank` overrides the number of doors inherited, uses the `start_up` method within the method `move`, and provides a new attribute. It therefore matches all three of our criteria.
- Similarly, the class `Car` overrides the number of doors and uses the method `start_up()`. It also uses the instance variable fuel within a new method `accelerate()`. It also, therefore, matches our criteria.
- The class `Dinghy` defines a new attribute `sails` and a new method `set_sail()`. As such, it does not use any of the features inherited from `Conveyance`. However, we might say that it has extended `Conveyance` by providing this attribute and method. We must then consider the features provided by `Conveyance`. We can ask ourselves whether they make sense within the context of `Dinghy`. If we assume that a dinghy is a small sail-powered boat, with no cabin and no engine, then nothing inherited from `Conveyance` is useful. In this case, it is likely that `Conveyance` is misnamed, as it defines some sort of a *motor vehicle*, and the `Dinghy` class should not have extended it.

20.7 Overriding Methods

Overriding occurs when a method is defined in a class (for example, `Person`) and also in one of its subclasses (for example, `Employee`). It means that instances of `Person` and `Employee` both respond to requests for this method to be run but each has their own implementation of the method.

For example, let us assume that we define the method `__str__()` in these classes (so that we have a string representation of these objects to use with the `print` function). The pseudo code definition of this in `Person` might be:

```
def __str__(self):
  return 'Person ' + self.name + ' is ' + str(self.age)
```

In `Employee`, it might be defined as:

```
def __str__(self):
  return 'Employee(' + str(self.id) + ')'
```

The method in `Employee` replaces the version in `Person` for all instances of `Employee`. If we ask an instance of `Employee` for the result of `__str__()`, we get the string '`Employee(<some_id>)`'. If you are confused, think of it this way:

> If you ask an object to perform some operation, then, to determine which version of the method is run, look in the class used to create the instance. If the method is not defined there, look in the class's parent. Keep doing this until you find a method which implements the operation requested. This is the version which is used.

As a concrete example, see the classes `Person` and `Employee` below; in which the `__str__()` method in `Person` is overridden in `Employee`.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def __str__(self):
        return self.name + ' is ' + str(self.age)

class Employee(Person):
    def __init__(self, name, age, id):
        super().__init__(name, age)
        self.id = id
    def __str__(self):
        return self.name + ' is ' + str(self.age) + ' - i
str(self.id) + ')'
```

Instances of these classes will both be convertible to a string using `__str__()` but the version used by instances of `Employee` will differ from that used with instances of `Person`, for example:

```
p = Person('John', 54)
print(p)
e = Employee('Denise', 51, 1234)
print(e)
```

Generates as output:

```
John is 54
Denise is 51 - id(1234)
```

As can be seen from this the `Employee` class prints the `name`, `age` and `id` of the `Employee` while the `Person` class only prints the `name` and `age`.

20.8 Extending Superclass Methods

However, in the previous section we had to duplicated the code in `Person` down in `Employee` so that we could convert the `name` and `age` attributes into strings.

However we can avoid this duplication by invoking the parent class's method from within the child class version (as we in fact did for the `__init__()` initialiser).

For example:

```
class Person:
   def __init__(self, name, age):
       self.name = name
       self.age = age

   def __str__(self):
     return self.name + ' is ' + str(self.age)

class Employee(Person):
   def __init__(self, name, age, id):
       super().__init__(name, age)
       self.id = id

   def __str__(self):
     return super().__str__() + '-id(' + str(self.id) + ')'
```

In this version of the code the `Employee` classes version of the `__str__()` method first calls the parent classes version of this method and then adds the location information to the string returned from that. This means that we only have one location that converts `name` and `age` into a string.

The output from the code

```
p = Person('John', 54)
print(p)
e = Employee('Denise', 51, 1234)
print(e)
```

remains exactly the same:

```
John is 54
Denise is 51 - id(1234)
```

20.9 Inheritance Oriented Naming Conventions

There are two naming conventions to be aware of with respect to Python classes and inheritance. These are that

- *Single underbar convention*. Methods or instance variables/attributes (those accessed via `self`) whose names start with a single under bar are considered to be *protected* that is they are private to the class but can be accessed from any subclass. Their scope is thus the class and any subclasses (either direct subclasses or any level of sub subclass).
- *Double underbar convention*. Method or instance variables/attributes (those accessed via `self`) whose names start with a double under bar should be considered *private* to that class and should not be called from outside of the class. This includes any subclasses; private means private to the class and only to that class.

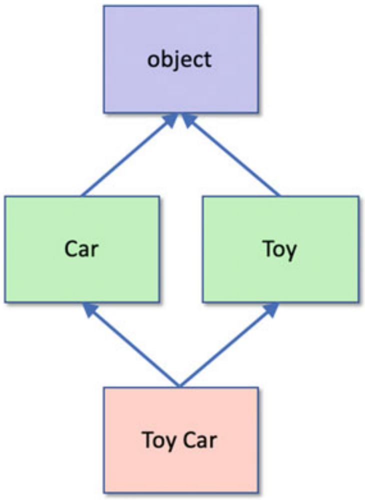
Any identifier of the form `__somename` (at least two leading underscores and at most one trailing underscore) is textually replaced with `_classname__somename`, where classname is the current class name with leading underscore(s) stripped.

Python does what is called *name mangling* to provide some support for methods that start with a double under bar. This mangling is done without regard to the syntactic position of the identifier, so it can be used to define class-private instance and class variables, methods, variables stored in globals, and even variables stored in instances.

20.10 Python and Multiple Inheritance

Python supports the idea of multiple inheritance; that is a class can inherit from one or more other classes (many object-oriented languages limit inheritance to a single class such as Java and C#).

This idea is illustrated by the following diagram:



In this case the class `ToyCar` inherits from the class `Car` and the class `Toy`. In turn the `Car` and `Toy` classes inherit from the (default) base class `object`.

The syntax for defining multiple inheritance in Python allows multiple superclasses to be listed in the parent class list (defined by the brackets following the class name). Each parent class is separated by a comma. The syntax is thus:

```
class SubClassName(BaseClassName1, BaseClassName2, …
BaseClassNameN):
    class-body
```

For example:

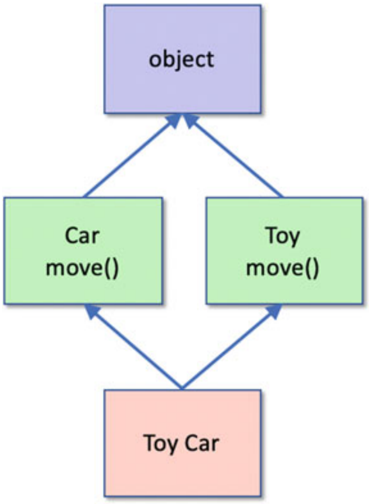
```
class Car:
    """ Car """

class Toy:
    """ Toy """

class ToyCar(Car, Toy):
    """ A Toy Car """
```

We can say that the class ToyCar inherits all the attributes (data) and methods (behaviour) defined in classes Car, Toy and object.

One of the fundamental questions that this raises is how is inheritance of behaviour managed within a multiple inheritance hierarchy. The challenge that multiple inheritance possesses is illustrated by adding a couple of methods to the class hierarchy we are looking at. In this example we have added the method move() to both the class Car and the class Toy:



The question here is which version of the method move() will be run when an instance of the ToyCar class is instantiated and we call toy_car.move()?

This illustrates (a simple version of) the so-called "diamond inheritance" problem.

The issue is that with multiple base classes from which attributes or methods may be inherited, there is often ambiguity that must be resolved. Here, when we create an instance of the class ToyCar, and call the move() method, does this invoke the one inherited from the Car base class or from the Toy base class?

The answer is that in Python 3, a *breadth first* search is used to find methods defined in parent classes; this means that when the method move() is called on ToyCar, it would first look in Car; it would then only look in Toy if it could not find a method move() in Car. If it cannot find the method in either Car or Toy it would then look in the class object.

As a result, it will find the version in Car first and use that version.

This is shown below:

```
class Car:
    def move(self):
        print('Car - move()')

class Toy:
    def move(self):
        print('Toy - move()')

class ToyCar(Car, Toy):
    """ A Toy Car """

tc = ToyCar()
tc.move()
```

The output of this is

```
Car - move()
```

However, if we alter the order in which the `ToyCar` inherits from the parent classes such that we swap `Toy` and `Car` around:

```
class ToyCar(Toy, Car):
    """ A Toy Car """
```

Then the `Toy` class is searched first and the output is changed to `Toy – move()`.

This shows that the order in which a class inherits from multiple classes *is significant* In Python.

20.11 Multiple Inheritance Considered Harmful

At first sight multiple inheritance in Python might appear to be particularly useful; after all it allows you to mix together multiple concepts into a single class very easily and quickly. This is certainly true and it can be a very flexible feature if used with care. However, the word *care* is used here and should be noted.

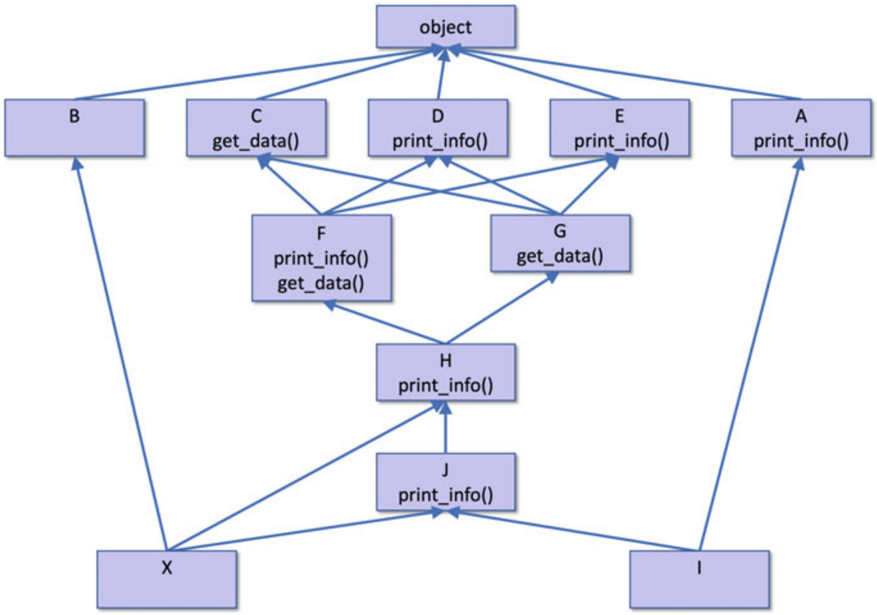
Multiple inheritance can also be very dangerous and is quiet a contentious topic for programmers and for those designing programming languages. Few things in programming are inherently bad but multiple inheritance can result in a level of complexity (and unexpected behaviour) that can tie developers in knots.

Part of the problem highlighted by those protesting against multiple inheritance is down to the increased complexity and ambiguity that can occur with multiple inheritance trees that may interconnect between the different classes. One way to think of this is that if a class inherits from multiple classes, then that class may have the same classes in the class hierarchy multiple times, this can make it hard to determine which version of a method may execute and this may allow bugs to go untouched or indeed introduce expected issues due to different interactions between methods. This is exacerbated when inherited methods call `super()` using the same method name such as:

```
def get_data(self):
    return super().get_data() + 'FData'
```

The following diagram presents a somewhat convoluted multiple inheritance example where the class names A-X have been used so that there is no semantic meaning attributable to the inherited classes. Different classes define several common methods (`print_info()` and `get_data()`).

All the classes in the hierarchy define a `__str__()` method that returns the class name; if the class extends a class other than object, then the super version of `__str__()` is also invoked:



The code for this class hierarchy is given at the end of the section to avoid breaking up the flow.

We can now use the `X` class in a simple Python program:

```
x = X()
print('print(x):', x)
print('-' * 25)
x.print_info()
```

The question now is what is the output from this program?

What is the string printed to represent `X`? What is printed out as a result of calling the method `print_info()`?

The output from this simple code is:

```
print(x): CGFHJX
-------------------------
HCDataGDataFData
```

However, if we change the order of inheritance for the class ‘`H`’ from `(F, G)` to `(G, F)` then the output changes:

```
print(x): CFGHJX
-------------------------
HCDataFDataGData
```

This is of course because the search order, back up through the class hierarchy, is now different.

Note that this change came about not because of a modification we made to the class we instantiated (that is the class `X`), but from the order of the classes that one of its parents inherited from. This can be one of the unintended consequences of multiple inheritance; changing something in the multiple class hierarchy at one level can break some behaviour further down the hierarchy in a class that is unknown to the developer.

Also note that the class inheritance diagram we presented earlier did not state what order the parent classes were listed in for any specific class (this was left to the discretion of the programmer).

Of course Python is not ambiguous nor does it get confused; it is the human developer that can get confused and be surprised with the behaviour that is then presented. Indeed, if you try and define a class hierarchy which Python cannot resolve into a consistent structure it will tell you so, for example:

```
Traceback (most recent call last):
  File "multiple_inheritance_example.py", line 65, in <mc
    class Z(H, J):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases F, G
```

What can be confusing is that Python's ability to produce a consistent structure can also be dependent on the order of inheritance. For example, if we modify the classes that 'X' inherits from such that the order is I and J:

```
class X(I, J):
    def __str__(self):
           return super().__str__() + 'X'
```

Then this compiles and can be used with the previous code (albeit with different output):

```
print(x): AIX
-------------------------
A
```

However, if we change the order of the parent classes such that we swap I and J:

```
class X(J, I):
    def __str__(self):
        return super().__str__() + 'X'
```

We now get a TypeError exception raised:

```
Traceback (most recent call last):
  File "multiple_inheritance_example.py", line 73, in <module>
    class X(J, I):
TypeError: Cannot create a consistent method resolution
order (MRO) for bases J, I
```

Therefore, in general care needs to be taken when utilising multiple inheritance; but that is not to say that such situations are not useful. In some cases you want a class to inherit from parents that have complete different hierarchies and are completely separate from each other; in such situations multiple inheritance can be very useful—these so called *orthogonal behaviours* are one of the best uses of multiple inheritance and should not be ignored merely due to concerns of increased complexity.

The class definitions used for the multiple inheritance class hierarchy are given below:

```python
class A:
    def __str__(self):
        return 'A'
    def print_info(self):
        print('A')

class B:
    def __str__(self):
        return 'B'

class C:
    def __str__(self):
        return 'C'
    def get_data(self):
        return 'CData'

class D:
    def __str__(self):
        return 'D'
    def print_info(self):
        print('D')

class E:
    def __str__(self):
        return 'E'
    def print_info(self):
        print('E')

class F(C, D, E):
    def __str__(self):
        return super().__str__() + 'F'
    def get_data(self):
        return super().get_data() + 'FData'
    def print_info(self):
        print('F' + self.get_data())

class G(C, D, E):
    def __str__(self):
        return super().__str__() + 'G'
    def get_data(self):
        return super().get_data() + 'GData'

class H(F, G):
    def __str__(self):
        return super().__str__() + 'H'
    def print_info(self):
        print('H' + self.get_data())
```

```
class J(H):
    def __str__(self):
        return super().__str__() + 'J'

class I(A, J):
    def __str__(self):
        return super().__str__() + 'I'

class X(J, H, B):
    def __str__(self):
        return super().__str__() + 'X'
```

20.12 Summary

To recap on the concept of inheritance. Inheritance is supported between classes in Python. For example, a class can extend (subclass) another class or a set of classes. A subclass inherits all the methods and attributes defined for the parent class(es) but may override these in the subclass.

In terms of the inheritance we say:

- A subclass inherits from a super class.
- A subclass obtains all code and attributes from the super class.
- A subclass can add new code and attributes.
- A subclass can override inherited code and attributes.
- A subclass can invoke inherited behaviour or access inherited attributes.

20.13 Online Resources

There are many resources available online relating to class inheritance including:

- https://docs.python.org/3/tutorial/classes.html#inheritance The Python software foundation tutorial on class inheritance.
- https://en.wikipedia.org/wiki/Multiple_inheritance which provides a discussion on multiple inheritance and the potential challenges it can introduce.

20.14 Exercises

The aim of these exercises is to extend the `Account` class you have been developing from the last two chapters by providing `DepositAccount`, `CurrentAccount` and `InvestmentAccount` subclasses.

Each of the classes should extend the `Account` class by:

- `CurrentAccount` adding an overdraft limit as well as redefining the withdraw method.
- `DepositAccount` by adding an interest rate.
- `InvestmentAccount` by adding an investment type attribute.

These features are discussed below:

The `CurrentAccount` class can have an `overdraft_limit` attribute. This can be set when an instance of a class is created and altered during the lifetime of the object. The overdraft limit should be included in the `__str__()` method used to convert the account into a string.

The `CurrentAccount` `withdraw()` method should verify that the balance never goes below the overdraft limit. If it does then the `withdraw()` method should not reduce the balance instead it should print out a warning message.

The `DepositAccount` should have an interest rate associated with it which is included when the account is converted to a string.

The `InvestmentAccount` will have a `investment_type` attribute which can hold a string such as 'safe' or 'high risk'.

This also means that it is no longer necessary to pass the type of account as a parameter—it is implicit in the type of class being created.

For example, given this code snippet:

```
#  CurrentAccount(account_number, account_holder,
#                 opening_balance, overdraft_limit)
acc1 = CurrentAccount('123', 'John', 10.05, 100.0)
#  DepositAccount(account_number, account_holder,
opening_balance,
#                 interest_rate)
acc2 = DepositAccount('345', 'John', 23.55, 0.5)
# InvestmentAccount(account_number, account_holder,
opening_balance,
#                   investment_type)
acc3 = InvestmentAccount('567', 'Phoebe', 12.45, 'high risk')

acc1.deposit(23.45)
acc1.withdraw(12.33)

print('balance:', acc1.get_balance())
acc1.withdraw(300.00)
print('balance:', acc1.get_balance())
```

Then the output might be:

```
balance: 21.17
Withdrawal would exceed your overdraft limit
balance: 21.17
```